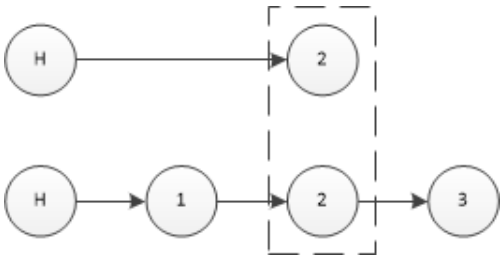
redis跳跃表图解&插入详述

跳跃表数据结构

```
typedef struct zskiplist {
    // 头节点，尾节点
    struct zskiplistNode *header, *tail;
    // 节点数量
    unsigned long length;
    // 目前表内节点的最大层数
    int level;
} zskiplist;
```

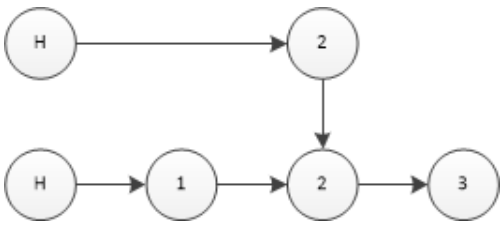
这里补充说一下节点个数，看下面这个图：



这个图里有几个节点呢？答案是3个，这里你可能有两个疑问：

为什么不算头节点呢？因为头节点不存储数据

那为什么两个2算一个节点呢？这个问题先记着，这里你看到我的图应该还会另一个问题，那就是我画的这个和你在其他地方看到的不一样，你看到的大多是这样的：



区别在于，我画的那个**并没有高层到低层的指针**，而是用一个虚线框把同一列框了起来，这又是为啥呢？接着看跳跃表节点的数据结构你就明白了。

跳跃表节点数据结构

还是先给源码：

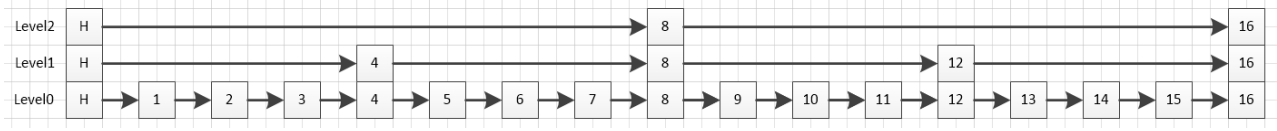
```
typedef struct zskiplistNode {
    // member 对象
    robj *obj;
    // 分值
    double score;
    // 后退指针
    struct zskiplistNode *backward;
    // 层
    struct zskiplistLevel {
        // 前进指针
        struct zskiplistNode *forward;
        // 这个层跨越的节点数量
        unsigned int span;
    } level[];
} zskiplistNode;
```

如果你对redis还不太熟，可能不理解里面的**分值**是干啥用的，这里先介绍一下，Redis跳跃表是用来实现有序集合（zset）的，zset就是集合里每一个成员都有一个对应的评分，成员是按评分从低到高存储的。因此，在redis跳跃表里，节点也是按分值从低到高排列的，而不是按对象本身的大小。

再解释一下后退指针，一个节点的后退指针会指向它的上一个节点，为什么需要后退指针呢？因为zset支持分数以从高到低的顺序返回集合元素，这个时候就会用到后退指针。

欧克，下面说重点，就是这个level数组，这可以解释上一节留下的两个问题，我们调回上面的图，首先，**在同一列的两个2，并不是两个节点，而是同一个节点里level数组**

的两个元素，其次，这两个2之间也没有指向关系，它们是通过存储在一个数组里，通过数组下标联系起来的，在进一步学习redis跳跃表之前，我认为搞明白这个是很有必要的。到这里，我们可以给出redis跳跃表的一种更严谨的画法（为了方便讲解后面的插入，这里画个更多节点）：



level呢，是一个结构体数组，结构体有两个成员，forward和span。

forward是指节点在这一层对应的下一个节点，换句话说，一个节点，在每一层都有不同的forward指针，就拿上图中的节点8来说，在level0，节点8的forward就是节点9，在level1，节点8的forward就是节点12，在level2，节点8的forward是16。

span呢，是指节点在这一层距离下一个节点的距离，这个变量可以用来快速的确定节点的排名，还拿上图中的节点8来说，在level0，节点8的span就是1，在level1，节点8的span就是4，在level2，节点8的span是8。

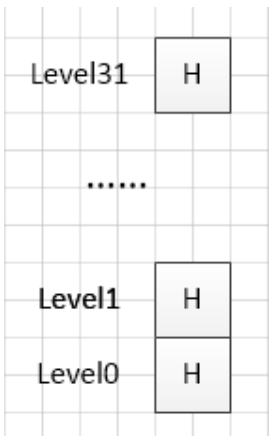
最后为了避免误会，我还想对我的图补充一句，16个节点插入以后跳跃表的样子并不会像我画的这样规整，节点在插入时，层数是随机的，对于一个节点，层数是n的概率是高于n+1的，所以当节点多了以后，会发现层数越高，节点越少。

跳跃表的创建

```
zskiplist *zslCreate(void) {
    int j;
    zskiplist *zsl;

    zsl = zmalloc(sizeof(*zsl));

    zsl->level = 1;
    zsl->length = 0;

    // 初始化头节点， O(1)
    zsl->header = zslCreateNode(ZSKIPLIST_MAXLEVEL,0,NULL);
    // 初始化层指针，O(1)
    for (j = 0; j < ZSKIPLIST_MAXLEVEL; j++) {
        zsl->header->level[j].forward = NULL;
        zsl->header->level[j].span = 0;
    }
    zsl->header->backward = NULL;

    zsl->tail = NULL;

    return zsl;
}
```

创建函数并不复杂，但还是说明几点。

- ZSKIPLIST_MAXLEVEL，这个是跳跃表的最大层数，源码里通过宏定义设置为了32，也就是说，节点再多，也不会超过32层。
- 初始化头节点，这里我们先看看初始化了头节点之后的初始跳跃表是什么样的：



也就是说，因为节点最多有32层，所以这里先把32层链表对应的头节点建立好。

其他的工作就是简单的初始化，这里就不说了。

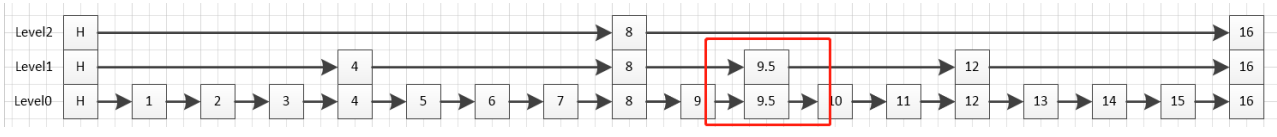
跳跃表的插入

```
zskiplistNode *zslInsert(zskiplist *zsl, double score, robj *obj) 登录后复制
    // 记录寻找元素过程中，每层能到达的最右节点
    zskiplistNode *update[ZSKIPLIST_MAXLEVEL], *x;

    // 记录寻找元素过程中，每层所跨越的节点数
    unsigned int rank[ZSKIPLIST_MAXLEVEL];

    int i, level;

    redisAssert(!isnan(score));
    x = zsl->header;
    // 记录沿途访问的节点，并计数 span 等属性
    // 平均 O(log N) ，最坏 O(N)
    for (i = zsl->level-1; i >= 0; i--) {
        /* store rank that is crossed to reach the insert position */
        rank[i] = i == (zsl->level-1) ? 0 : rank[i+1];

        // 右节点不为空
        while (x->level[i].forward &&
            // 右节点的 score 比给定 score 小
            (x->level[i].forward->score < score ||
                // 右节点的 score 相同，但节点的 member 比输入 member 要小
                (x->level[i].forward->score == score &&

                 compareStringObjects(x->level[i].forward->obj,obj) < 0))) {
            // 记录跨越了多少个元素27
         rank[i] += x->level[i].span;28              // 继续向右前进
            x = x->level[i].forward;
        }
        // 保存访问节点
        update[i] = x;
    }

    /* we assume the key is not already inside, since we allow duplicated

 * scores, and the re-insertion of score and redis object should never

 * happpen since the caller of zslInsert() should test in the hash table
     * if the element is already inside or not. */39
// 因为这个函数不可能处理两个元素的 member 和 score 都相同的情况，40
// 所以直接创建新节点，不用检查存在性41   42      // 计算新的随机层数
    level = zslRandomLevel();
    // 如果 level 比当前 skiplist 的最大层数还要大
    // 那么更新 zsl->level 参数
    // 并且初始化 update 和 rank 参数在相应的层的数据
    if (level > zsl->level) {
        for (i = zsl->level; i < level; i++) {
            rank[i] = 0;
            update[i] = zsl->header;
            update[i]->level[i].span = zsl->length;
        }
        zsl->level = level;
    }

    // 创建新节点
    x = zslCreateNode(level,score,obj);
    // 根据 update 和 rank 两个数组的资料，初始化新节点
    // 并设置相应的指针
    // O(N)
    for (i = 0; i < level; i++) {
        // 设置指针
        x->level[i].forward = update[i]->level[i].forward;
        update[i]->level[i].forward = x;

        /* update span covered by update[i] as x is inserted here */
        // 设置 span

     x->level[i].span = update[i]->level[i].span - (rank[0] - rank[i]);
        update[i]->level[i].span = (rank[0] - rank[i]) + 1;70     }

    /* increment span for untouched levels */
    // 更新沿途访问节点的 span 值
    for (i = level; i < zsl->level; i++) {
        update[i]->level[i].span++;
    }
```

```
78      // 设置后退指针79
    x->backward = (update[0] == zsl->header) ? NULL : update[0];80
    // 设置 x 的前进指针81      if (x->level[0].forward)
82          x->level[0].forward->backward = x;
83      else
84          // 这个是新的表尾节点
85          zsl->tail = x;
86
87      // 更新跳跃表节点数量
88      zsl->length++;
89
90      return x;
91  }
```

思路

首先上面说过，新加入的节点，层数是随机的，我们先不管怎么随机，就假设要在上面那个图上插入一个节点，分值是9.5，且随机生成的层数是2，那么插入这个节点之后，跳跃表应该是这样的：



如果要实现这个效果，我们要分两步去做：

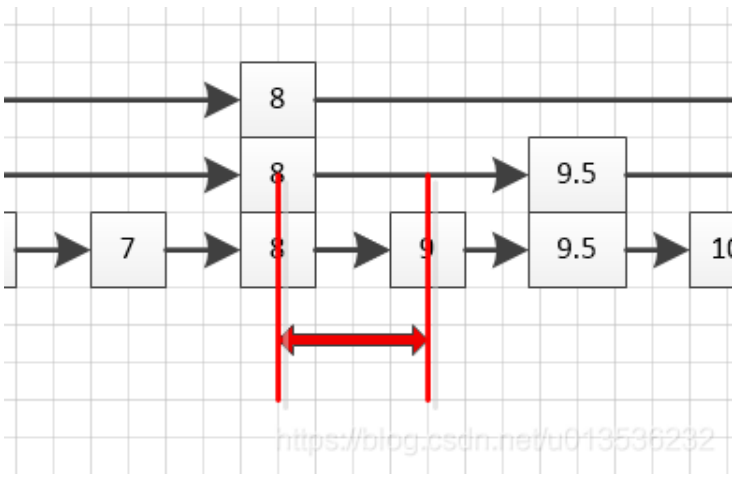
- 找到新节点在每一层的上一个节点（即：对于level0，应该先找到节点9；对于level1，应该先找到节点8）
- 将新节点插入到每一层的上一个节点和下一个节点之间

除此之外，不要忘了每一层还有一个span变量，在插入新节点之后，我们需要计算新节点在每一层的span，另外，在插入新节点之后，新节点的上一个节点的span也会发生变化，需要我们更新。那么问题来了，这个span怎么来计算呢?

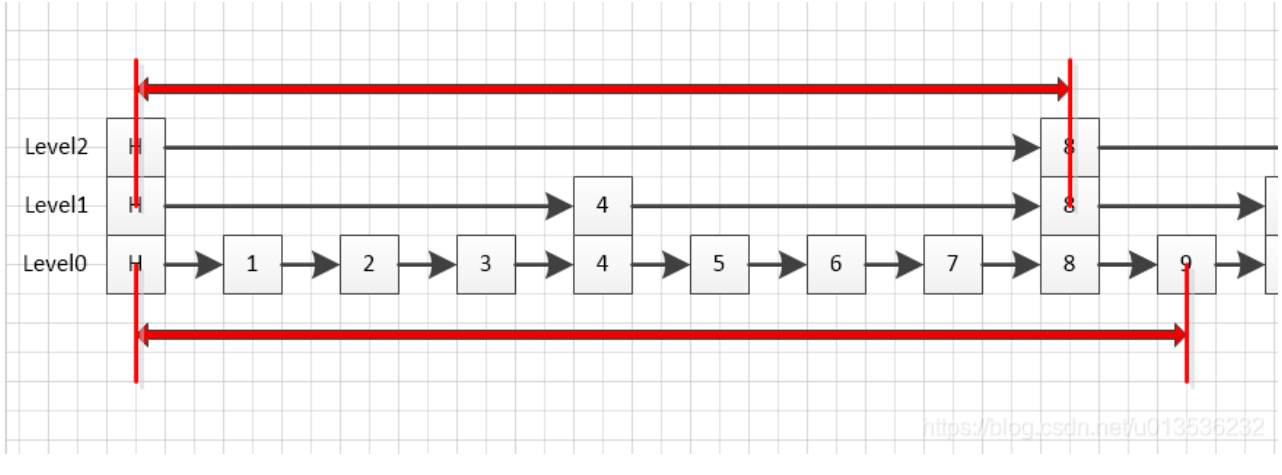
首先看底层，底层其实是没有这个问题的，因为底层压根就没有跳嘛。。。所有节点在底层都是直接相邻的，所以span都是1。

问题在于高层，比如level1，我们要想办法确定插入节点9.5之后，节点8到节点9.5之间的距离，和节点9.5到节点12之间的距离，实际上这两个确定一个就可以，因为这两个距离之和就是原来level1层节点8的span+1（+1是因为插入了节点9.5），既然如此，我们就想办法确定节点8到节点9.5之间的距离。

节点到节点9.5之间的距离又该怎么算呢？我们可以将这段距离转化为level1的节点8到level0的节点9之间的距离+1，也就是下面这段：



而这一段又该怎么算呢？可以把它当做下面两段距离的差：



而这个距离，其实就是当前节点的排名（节点8不就是第8个节点吗），同时也是当前节点之前，每一个节点的span之和（在level2，头节点的span是4，节点4的span是4，

加起来就是8），到这里，你是不是已经有点豁然大明白的感觉了呢，要是再不明白，我也没法再细说了，因为再细就是代码了，所以接下来就直接看代码吧。

遍历，记录update和rank

对应的是下面这一段：

```
    // 记录寻找元素过程中，每层能到达的最右节点
    zskiplistNode *update[ZSKIPLIST_MAXLEVEL], *x;

    // 记录寻找元素过程中，每层所跨越的节点数
    unsigned int rank[ZSKIPLIST_MAXLEVEL];

    int i, level;

    redisAssert(!isnan(score));
    x = zsl->header;
    // 记录沿途访问的节点，并计数 span 等属性
    // 平均 O(log N) ，最坏 O(N)
    for (i = zsl->level-1; i >= 0; i--) {
        /* store rank that is crossed to reach the insert position */
        rank[i] = i == (zsl->level-1) ? 0 : rank[i+1];

        // 右节点不为空
        while (x->level[i].forward &&
            // 右节点的 score 比给定 score 小
            (x->level[i].forward->score < score ||
                // 右节点的 score 相同，但节点的 member 比输入 member 要小
                (x->level[i].forward->score == score &&

                 compareStringObjects(x->level[i].forward->obj,obj) < 0))) {
            // 记录跨越了多少个元素25
          rank[i] += x->level[i].span;26              // 继续向右前进
            x = x->level[i].forward;
        }
        // 保存访问节点
        update[i] = x;
    }
```

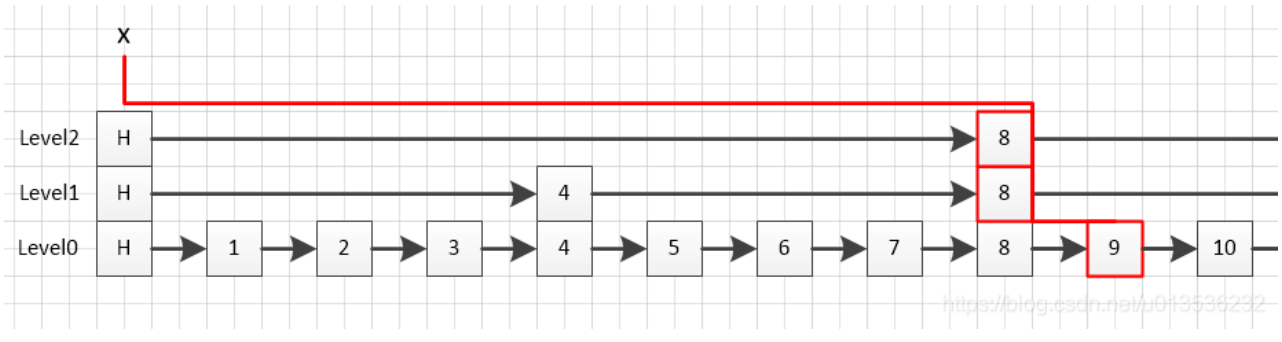
首先，这里创建了两个数组，数组大小都是最大层数，其中：

- **update数组**用来记录新节点在每一层的上一个节点，也就是新节点要插到哪个节点后面；
- **rank数组**用来记录update节点的排名，也就是在这一层，update节点到头节点的距离，这个上一节说过，是为了用来计算span。

update和rank数组的值可以通过一次逐层的遍历确定。

遍历之前，定义一个x指向头节点。

遍历的过程可以参照这个图来看，还是假设要插入9.5，只不过这时候我们还不知道随机生成的层数是多少，也就是说每一层都有可能插入，因此从跳跃表当前的最大层数开始遍历，遍历到最底层为止。



遍历过程中，如果x的forward指针指向的节点（也就是x的下一个节点）的评分低于插入节点的评分，那么插入节点应当插入x的下一个节点的右侧，所以这时rank应当加上x节点的span（也就是x到x下一个节点的距离），然后再将x指向x的下一个节点。这就是下面这段代码的含义。

```
        while (x->level[i].forward &&
            // 右节点的 score 比给定 score 小
            (x->level[i].forward->score < score ||
                // 右节点的 score 相同，但节点的 member 比输入 member 要小
                (x->level[i].forward->score == score &&

```

```
                compareStringObjects(x->level[i].forward->obj,obj) < 0))) {
7               // 记录跨越了多少个元素 8
            rank[i] += x->level[i].span; 9              // 继续向右前进
                                                                    10
            x = x->level[i].forward;11          }
```

上面这段循环的退出条件有两个，一个是x节点的下一个节点是空，也就是走到结尾了；另一个是x节点的下一个节点的评分大于插入节点的评分。在这两种条件下，都说明，新节点就是要插入在x节点后面，所以此时应将这一层的update节点记录为当前的x节点。

```
update[i] = x;
```

第一次遍历，就是level2，找到的update节点是节点8，对应的rank是此时x节点的span，也就是头节点在level2的span。x这时也移动到了节点8。

第二次遍历，注意这里不是再重头开始遍历了，因为上一次遍历完，x已经移到了level2的update节点，而在level1，update节点一定在x当前的位置之后（>=），所以对于rank的计算，也可以直接在上一层的rank的基础上继续计算，这就是下面这行代码的含义。

```
rank[i] = i == (zsl->level-1) ? 0 : rank[i+1];
```

生成随机层数

redis的跳跃表在插入节点时，会随机生成节点的层数，通过控制每一层的概率，控制每一层的节点个数，也就是保证第一层的节点个数，之后逐层增加，下面给出随机层数的生成代码：

```
1  int zslRandomLevel(void) {
2      int level = 1;
3      while ((random()&0xFFFF) < (ZSKIPLIST_P * 0xFFFF))
4          level += 1;
5      return (level<ZSKIPLIST_MAXLEVEL) ? level : ZSKIPLIST_MAXLEVEL;
6  }
```

这里面有一个宏定义ZSKIPLIST_P ，在源码中定义为了0.25，所以，上面这段代码，生成n+1的概率是生成n的概率的4倍。

如果生成的层数比当前跳跃表层数大

所以如果生成了一个比当前最大层数大的数，那么多出来的那些层也需要插入新的节点，而上面的那次遍历是从当前跳跃表最大层数开始的，也就是多出来这些层的update节点和rank还没有获取，因此需要通过下面这段程序，给多出来的这些层写入对应的rank和update节点。这部分很简单，因为这些层还没有节点，所以这些层的update节点只能是头节点，rank也都是0（头节点到头节点），而span则是节点个数（本身该层的头节点此时还没有forward节点，也不该有span，但插入节点后新节点需要用这个span计算新节点的span，因此这里需要把span设置为当前跳跃表中的节点个数）。

```
1      if (level > zsl->level) {
2          for (i = zsl->level; i < level; i++) {
3              rank[i] = 0;
4              update[i] = zsl->header;
5              update[i]->level[i].span = zsl->length;
6          }
7          zsl->level = level;
8      }
```

插入新节点

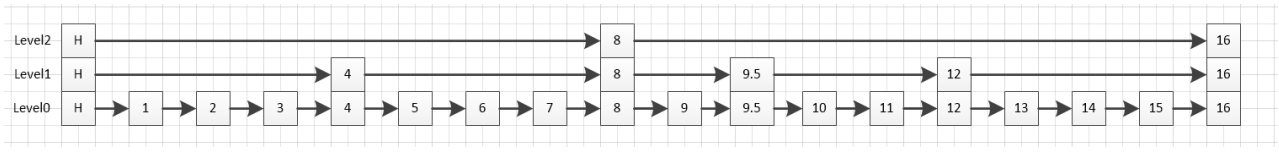
前面已经找到插入位置（update）了，接下来的插入其实就是单链表插入，这个就不说了。

注意span的计算，这里可以对着上面思路那一节来看， (rank[0] - rank[i])就是上面说的两段距离之差。

```
    x = zslCreateNode(level,score,obj);
    for (i = 0; i < level; i++) {
        // 设置指针
        x->level[i].forward = update[i]->level[i].forward;
        update[i]->level[i].forward = x;

        /* update span covered by update[i] as x is inserted here */
        // 设置 span

     x->level[i].span = update[i]->level[i].span - (rank[0] - rank[i]);
        update[i]->level[i].span = (rank[0] - rank[i]) + 1;11     }
```

更新未涉及到的层

如果随机生成的层数小于之前跳跃表中的层数，那么大于随机生成的层数的那些层在创建新节点的过程中就没有被操作到（创建新节点的时候是从0遍历到随机生成的层数），对于这些没有操作到的层，里面的update节点对应的span应当+1（因为后面插入了一个节点）。



还是拿插入9.5这个节点来说，如果插入过程中生成的随机层数是2，那么在插入新节点那一段程序中，只会更新level1中节点8的span和level0中节点9的span，而level中节点8的span也是需要+1的，所以我们需要手动更新一下为涉及到的层。

下面这段代码做的就是这件事。

```
    for (i = level; i < zsl->level; i++) {
        update[i]->level[i].span++;
    }
```

设置后继指针

针对每一层的调整到这里已经全部完成了，也就是level数组已经搞定，接下来，处理一下backward指针，首先新节点的backward要指向前一个节点，然后，新节点的下一个节点要将backward指向新节点。

```
    x->backward = (update[0] == zsl->header) ? NULL : update[0];
    // 设置 x 的前进指针
    if (x->level[0].forward)
        x->level[0].forward->backward = x;
    else
        // 这个是新的表尾节点
        zsl->tail = x;
```

更新跳跃表节点个数

最后，全部搞定，把跳跃表个数加一就大功告成了！

```
  zsl->length++;
```